CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science

4287:TR:81

Computational Arrays for Band Matrix Equations

by

Lennart Johnsson

1. Introduction

An important class of systems of linear equations is characterized by the matrix being of band type. The band may be naturally related to the definition of the problem or may be the result of transformations of the original problem. For instance, Householder Transformations can be used to reduce a symmetric matrix to three-diagonal form.

A band matrix is a matrix where the elements $a_{ij} = 0$ for $j>i+s$ and $i>j+r$. The bandwidth b is defined as r+s+1. The arrays that are proposed here are of particular interest when the bandwidth is small compared to the size of the matrix.

Gaussian elimination is a common and efficient method for solving linear vsystems of equations, especially when pivoting is unnecessary. The arrays studied here are based on Gaussian elimination as the method of solution. Gaussian elimination can proceed in several ways, for instance row by row or column by column. It is well-known that operations at the vector level can be performed concurrently. This form of concurrency is explored in the hexagonal arrays by H T Kung and Charles E. Leiserson [4] for band matrix factorization (and multiplication) and the orthogonal arrays by S Y Kung [5] for full matrix problems. For sparse matrix problems a higher degree of concurrency is possible. A three-diagonal problem can be solved in $\log_2 N$ steps, Johnsson [3]. The two-dimensional arrays we present here solve a system of equations in O(N) time using r*s multiply/add cells. The arrays are tailored to the communication requirements of Gaussian elimination as are the arrays by H T Kung and S Y Kung. Hence, the topology of the arrays are essentially the same. The discussion here focuses on

- techniques to avoid broadcasting of data and control

- control and data organizations required to be able to use arrays of a given bandwidth for the solution of systems of equations of a larger bandwidth.

Columnwise Gaussian elimination without pivoting proceeds as follows. To eliminate the nonzero elements in the first column the first row multiplied by suitable factors is subtracted from all other rows. The factors are determined so that the elements in the first column become zero. To eliminate the nonzero elements below the diagonal in the second column the same steps are repeated using the second row instead of the first. This procedure is repeated until all elements below the diagonal is eliminated. All multiplications and subtractions involved to make the elements below the diagonal in a column equal to zero can be performed concurrently. For the first column $(N-1)^2$ operations can be performed concurrently. This figure decreases as the elimination proceeds. In a band matrix there are only r*s nontrivial operations for the first N-s columns.

In the design of computational networks a trade-off between area and time or performance is an important design issue. The trade-off is strongly dependent on the organization in time and space of the input data in the computational problem studied here. The matrix coefficients are assumed to enter the network in a number of input streams. In the basic forms of the networks that are presented one input stream is associated with every diagonal of the matrix. One input stream is also associated with every right-hand side of the system of equations to be solved.

The concept of data streams is quite useful in gaining an understanding of the basic behavior of computational networks, S Y Kung [5], Weiser and Davis [6]. A concept closely related to that of data streams is that of wave fronts. A wave front illustrates the synchronization between data streams. Wave fronts are made up of points from different data streams and at most one point from one stream. A wave front in physics is a set of points having the same phase. The same property can also be used in determining wave fronts in a computational network. In the networks we vstudy there are several different data streams. Wave fronts are naturally associated with the set of data streams that corresponds to a set of data of some common characteristics. Such characteristics may, for example, be that the streams carry data from the right-hand side of the equations, the matrix defining the set of equations, the upper or lower triangular matrix, etc.

The computational arrays described here are not derived in a formal way as the arrays described in Johnsson et al [2]. The correctness of the arrays can, however, be verified in that manner, even though it is not carried out here. We use the same notation for storage elements as was used by Cohen in [1]. The modeling of storage is a key element in the formal approach to the design and verification of computational networks.

In this study the focus is on the data organization in space and time, and the synchronization between data streams at the word level. Implementations can be made using either bit serial or parallel representation of words. Symbols denoting storage, adders, multipliers and data paths should be interpreted as dealing with words. The models and notation used here can also be used in the further treatment of the arrays at the bit level.

In the array we discuss first, all computations connected with the elimination of the r elements within the band below the diagonal are carried out concurrently. In the naive form of the array the row that is subtracted from the other rows after having been multiplied by the proper factors is broadcasted through the array columnwise. The factors are broadcasted rowwise. The forward solution on the right-hand side of the equations is performed concurrently with the elimination on the matrix. The backward solution is performed after the elimination phase is completed.

For large values of r and s the naive form of the network will necessarily be slow due to long wires. Pipelining can be used to eliminate broadcasting and increase the computational rate. A small amount of additional hardware is required. In the pipelined version of the array we still associate an input stream with each nonzero diagonal of the matrix. Pipelining can be introduced in several ways. The way in which the data streams interact cause some information to traverse the network in loops. The data flow can be considered to be turbulent. The information is changed as it traverses the loop. Data from different data streams has to meet at the right place at the right time in order for the arrays to produce a correct output. Pipelining has several consequences upon the synchronization between data streams and the flow rate within a stream. Pipelining causes data streams to be delayed or skewed with respect to each other. The loops require data in a data stream to be spaced out in time if pipelining is used. The spacing out of data can be used to reduce the pin requirements by multiplexing a few data streams into one.

Pivoting can be accomplished with relative ease in the naive array. In the pipelined version pivoting cannot be accomplished without a performance penalty and complex

control. The flow of data is no longer stationary. The direction of the flow changes according to the selection of the pivot element. Also, the flow is not continuous in fully pipelined arrays that perform partial pivoting. There are waves of computational activity that perpetuates throughout the array. The time and location of the origin of the waves is determined by the location of the pivot elements. The distance between consecutive waves is, in general, not constant.

If the value of r or s is greater than what can be accommodated within the array, some form of mapping or partitioning of the problem has to take place. Time has to be substituted for space. The organization of the input data becomes more complex. The control of the data flow that was implicit in the interconnection scheme of the modules in the arrays of sufficient size now has to be made explicit. The computations in the alternatives discussed here proceed in phases. The flow of data is not the same in all phases. In the first alternative presented the phases can be thought of as defined by the change in direction of the data flow. In the first alternative, storage of intermediate results are made internal to the array. The second alternative is based on partitioning the original problem into blocks that are treated one at a time within the array. Intermediate results are stored externally to the array. The first alternative can also be thought of as a memory sparsely populated by processing capabilities. A few ways to organize the data and the computations in such a "smart" memory is described. In the second alternative an arithmetic processing capability is associated with a set of storage locations within the array that needs it combined with a regular storage without processing capability outside the array.

The organization of the computational elements and the data flow is tailored to the communication needs of Gaussian elimination. S Y Kung's [5] array for the solution of systems of equations characterized by a full matrix is also based on Gaussian elimination as was mentioned above. Hence, a great deal of similarity can be found between the two-dimensional arrays we propose here for band matrix equations and the array by S Y Kung. However, the mode of operation of the arrays proposed here is different since only a fraction of the matrix is present within the array at any one time. For Gaussian elimination with partial pivoting we also propose a linear array that have the same performance as the two-dimensional array. Data enters and leaves the array presented here in a continuous fashion. The arrays proposed here are used to study pipelining in two-dimensional arrays and its effect on data organization, data acceptance rate, and computational rate. The problem of using an array of a given size for the computation of the solution to a system of equations of a larger bandwidth than the array can treat concurrently was not treated by Kung [5].

The arrays are discussed informally in this report. The notation proposed by Johnsson et al. [2] can be used to formally verify the correctness of the arrays.

2. Arrays assuming no pivoting

Gaussian elimination without partial pivoting is studied first. An input channel is associated with each diagonal of the band matrix. The implementations concurrently performs the r*s operations involved in eliminating the nonzero elements in a column. The forward solution can be carried out concurrently as in the implementation in

Figure 1. For the concurrent solution of sets of equations with several right members arrays for the columns of the right members can be added. The data that enters the array during a cycle is indicated in Figure 2. The contents of the stacks after the first phase of the computations is shown in Figure 3.

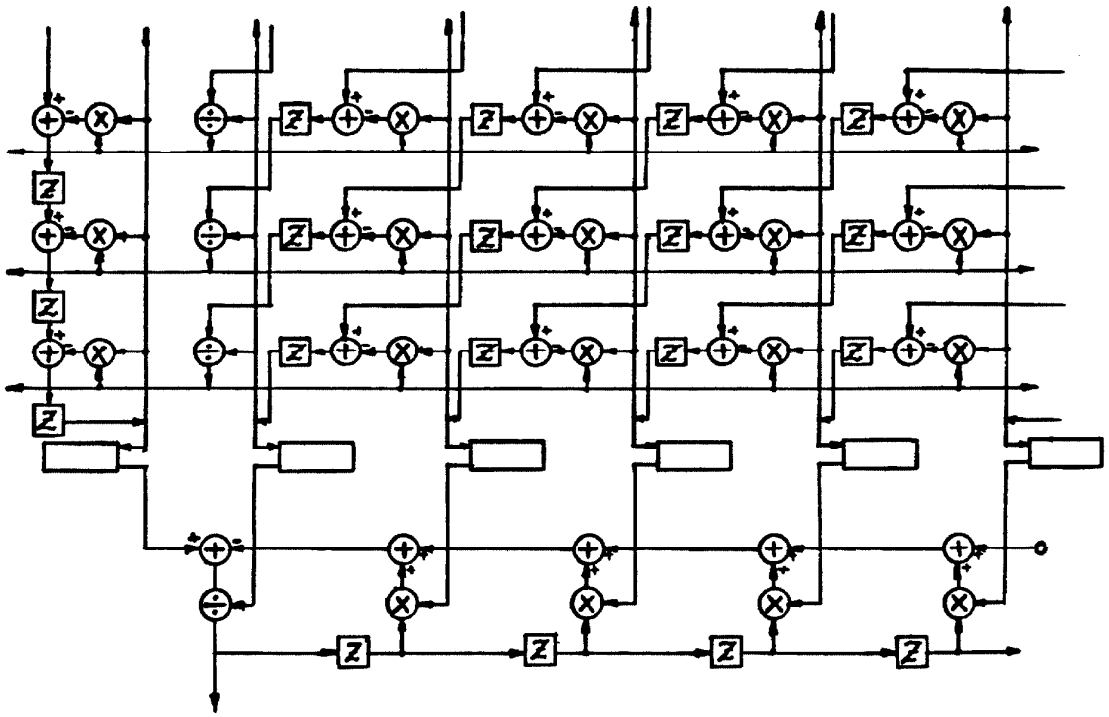


Figure 1. Array performing Gaussian elimination on band matrices

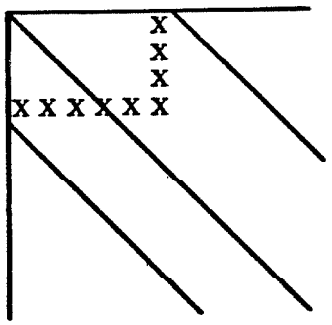


Figure 2. Concurrent input data

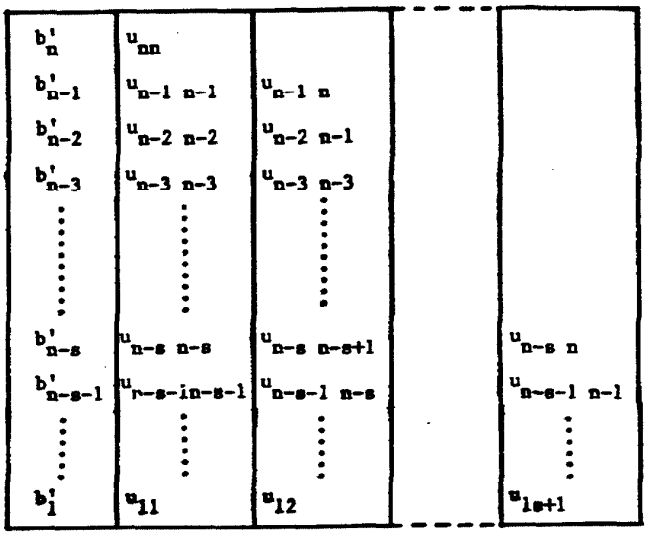


Figure 3. Stack content

The array in Figure 1 has r*s multiply add cells, b input channels for the matrix coefficients, one input channel for each of the right members and one output channel for each solution vector. The number of cycles required to compute a solution is 2N+r-1. If several problems are to be solved pipelining can be used to save r-1 cycles. From Figure 1 it is apparent that data is broadcasted both horizontally and vertically throughout the array. The coefficients in the pivot row are broadcasted vertically. The factors with which the pivot row is to be multiplied to eliminate the nonzero elements in the column treated are broadcasted horizontally. Broadcasting in the horizontal direction can be avoided by introducing delays between each column of cells in the array. To preserve correct operation it is necessary that the proper data items meet in the proper place at the right time. A correctly working array with no horizontal broadcasting is shown in Figure 4. The data entering the array at the same time corresponds to the part of a column falling within the band Figure, 5. The input data streams corresponding to diagonals above the main diagonal can be thought of as being delayed or skewed one step with respect to each other. The data within the stacks after the first phase is shown in Figure 6.

An implementation according to Figure 4 will, however, have a computational rate that is even lower than the array in Figure 1 due to the fact that s adders are connected in series in addition to long wires. There are also r multipliers that have to be driven in parallel. The computational rate can be improved by reintroducing the delays along the diagonals, Figure 7.

The implementation in Figure 7 is likely to have a much improved computational rate for a large array compared to the previous alternatives. The input streams to the array in Figure 7 can only contain valid data every second cycle. This is due to the fact that there are two delays in the shortest loops that data has to traverse.

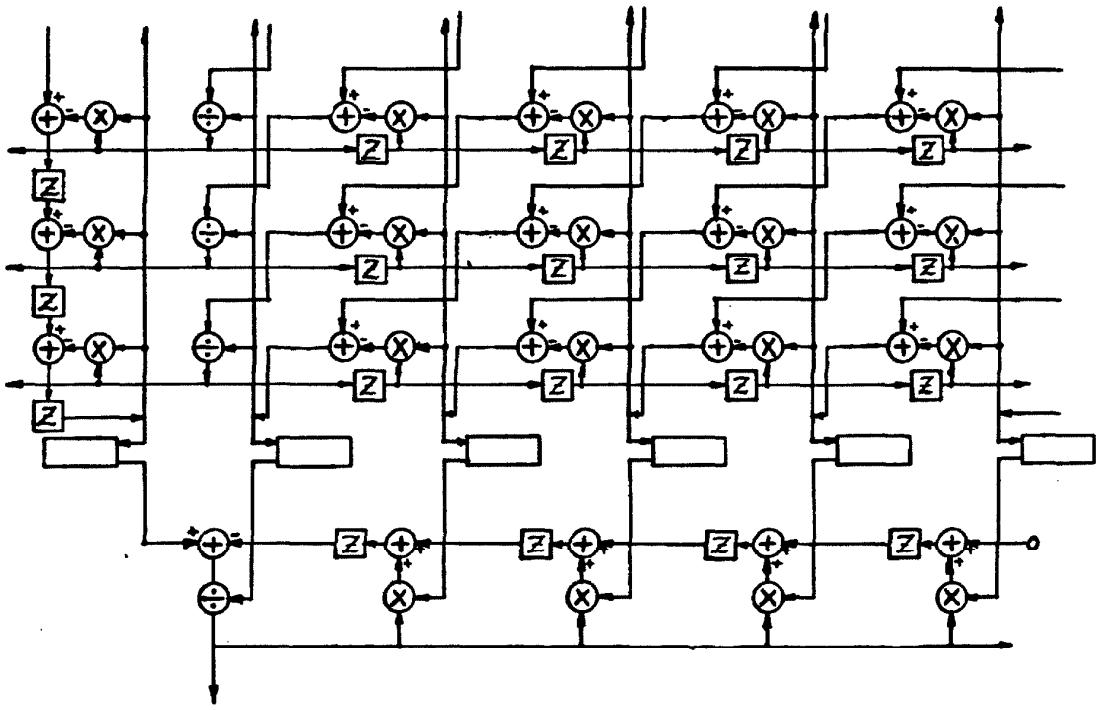


Figure 4. Arrays with no horizontal broadcasting

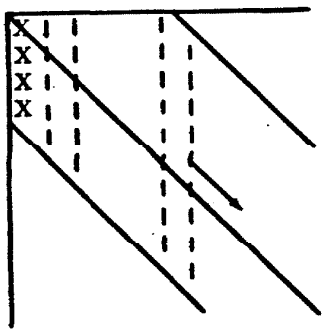


Figure 5. Concurrent input data

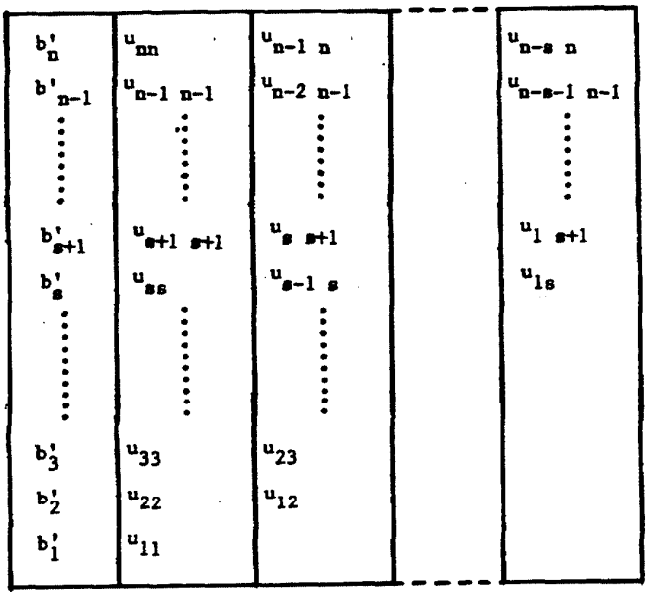


Figure 6. Stack content

The set of data being entered at any given time is shown in Figures 8a and 8b. The skewing between the data streams should be apparent from Figure 8b.

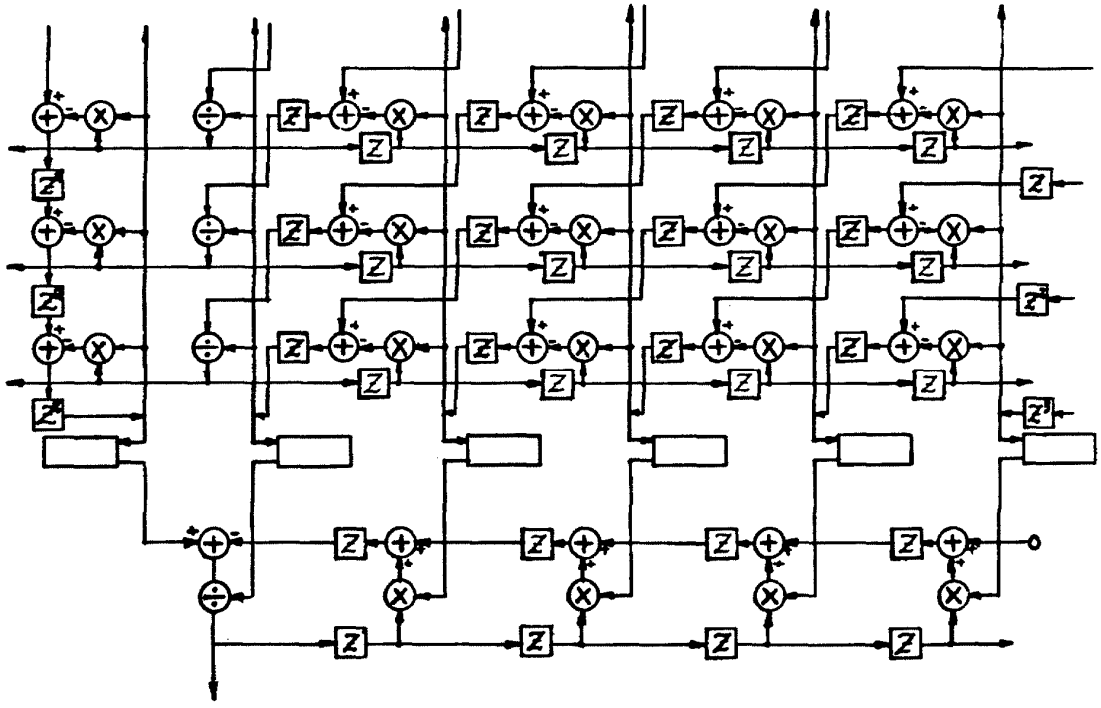


Figure 7. Pipelined input, no horizontal broadcasting

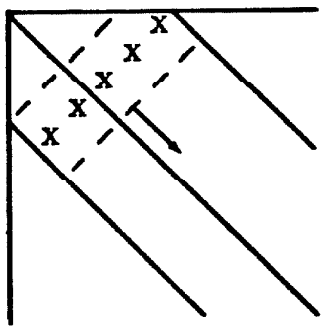


Figure 8a. Concurrent input data

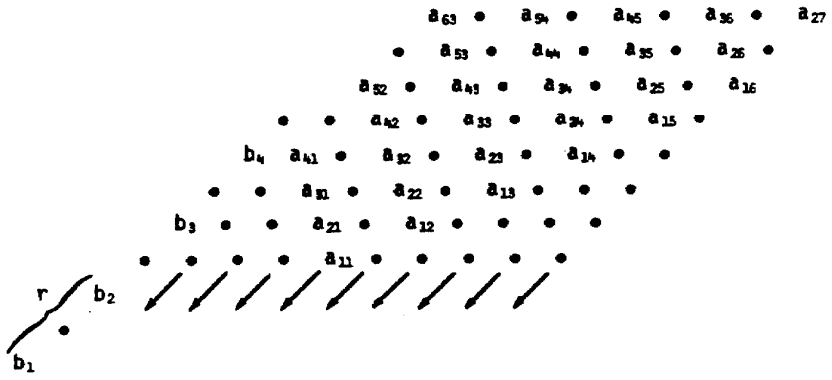


Figure 8b. Synchronization between data streams and data spacing within streams

The content of the stacks after the first phase is shown in Figure 9. In order to save stack space a push onto the stack can be executed every other cycle. Another alternative with the same effect is to combine two stacks into one. Combining two stacks into one can be accomplished by simply multiplexing outputs from two neighboring columns into one stack since data in neighboring columns become available at successive cycles.

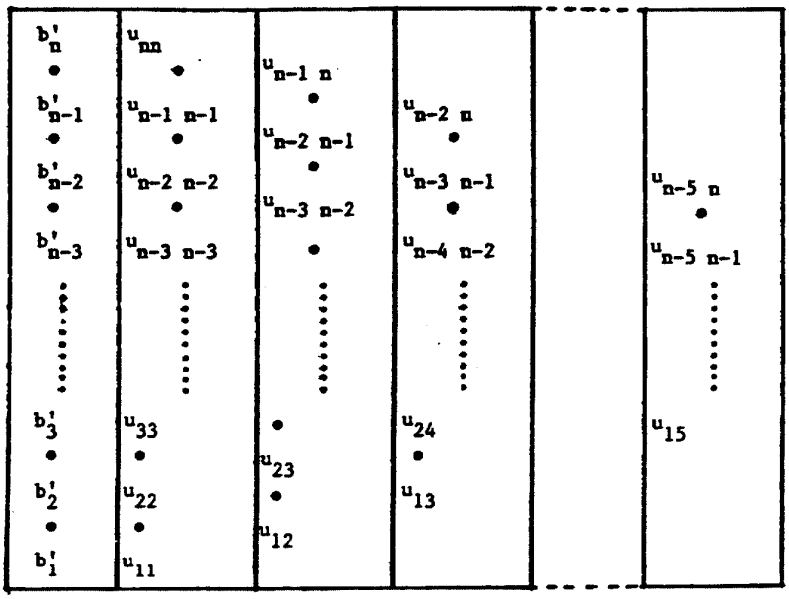


Figure 9. Stack content without any stack size reduction technique.

The broadcasting along the vertical wires can be treated similarly to the broadcasting on the horizontal wires. Figure 10 shows an implementation with no broadcasting, but with the adders on the diagonals of the arrays connected in series. The data organization is the same as in Figures 8 and 9. In this case the delays in the vertical direction requires the data streams corresponding to diagonals below the main diagonal to be delayed with respect to each other and the main diagonal.

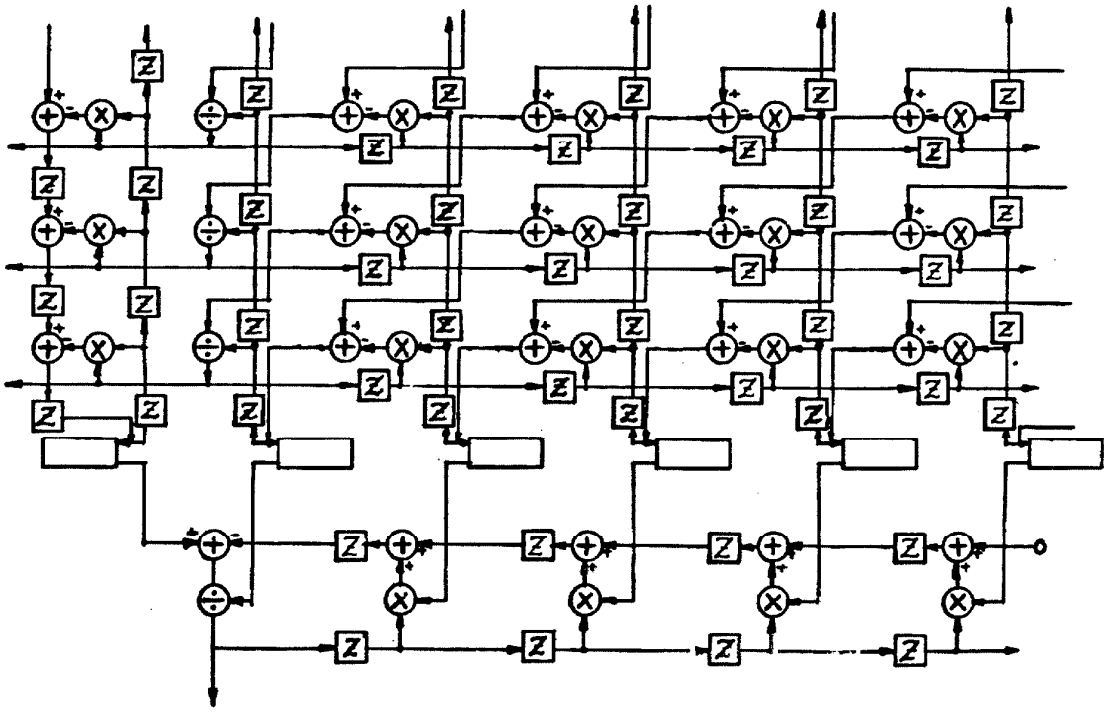


Figure 10. No horizontal and no vertical broadcasting.

The implementation in Figure 10 will be slow since several computational elements are connected in series. It is desirable to reintroduce the delays on the diagonals to improve the computational rate. The price paid is that only every third value in the input streams is significant. This is due to the fact that the minimum loops now contain three delays. Figure 11 shows an implementation which should have good computational rate. Figure 12 illustrates the corresponding data organization and Figure 13 the content of the stacks after the initial phase.

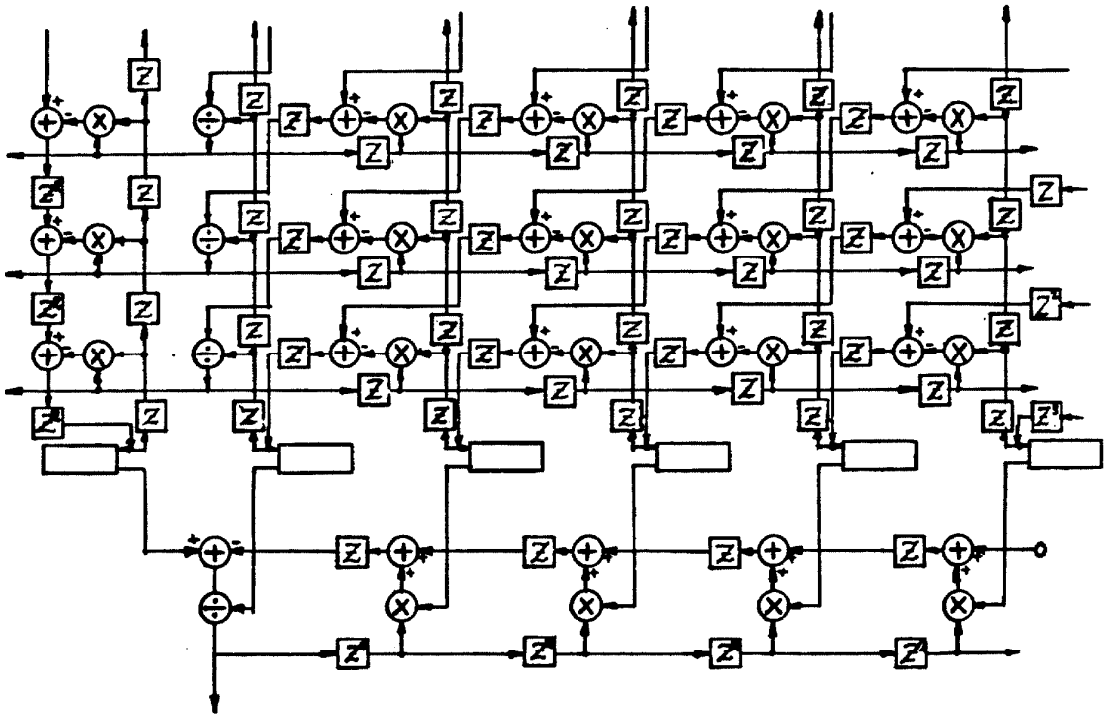


Figure 11. A fully pipelined array.

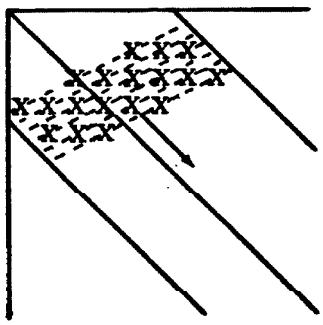


Figure 12a. Sets of concurrent input data.

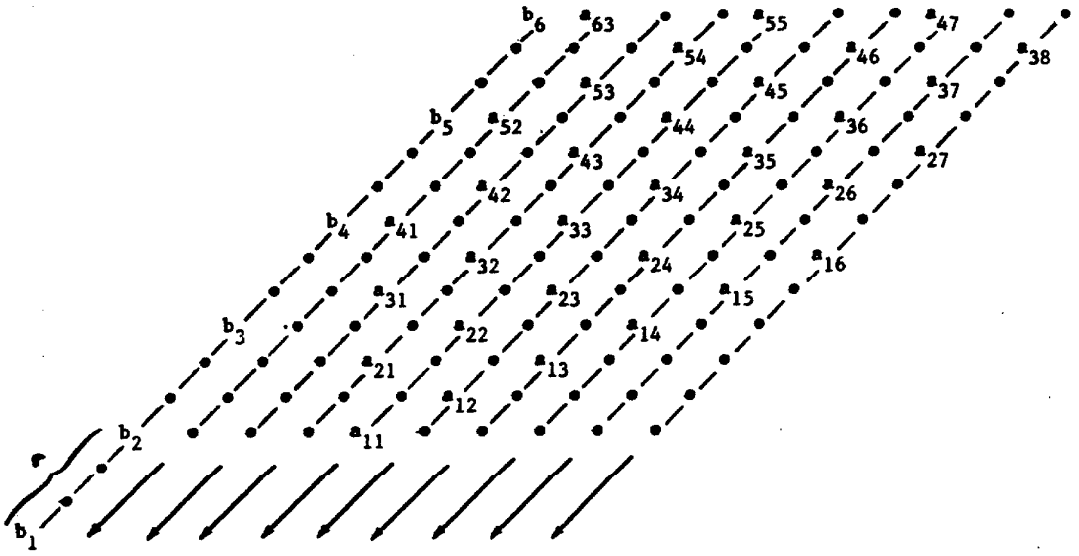


Figure 12b. Synchronization between input streams and data spacing within streams.

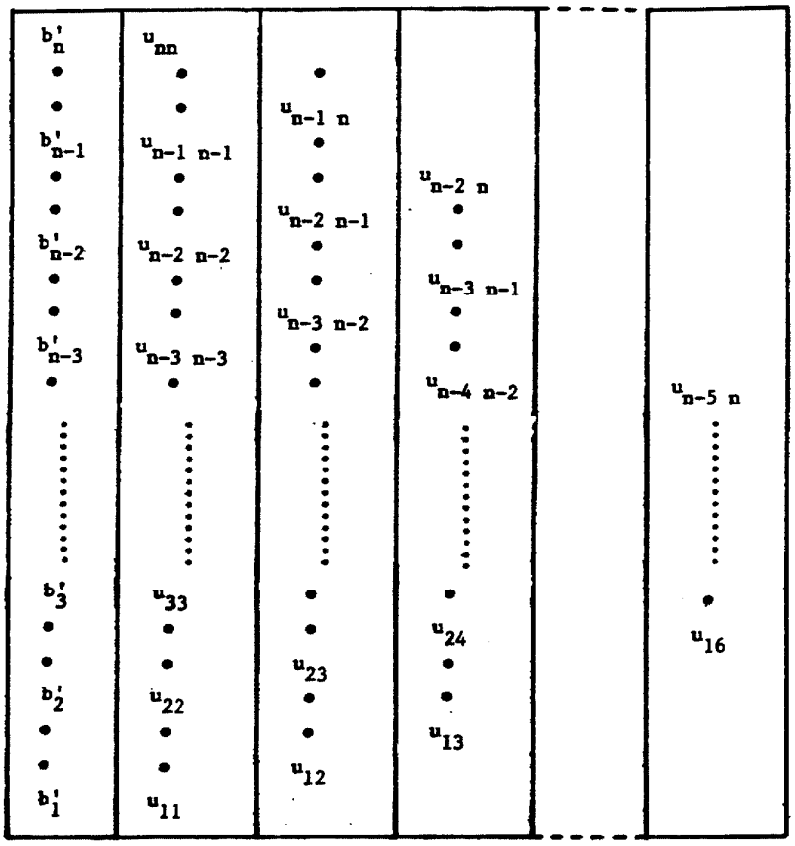


Figure 13. Stack content with no stack size reduction technique.

The implementation shown in Figure 11 can be considered as a fully pipelined version of the implementation in Figure 1. Hence pipelining improves the computational rate but input data in any input stream shall appear only every third cycle. Data in the different input streams are, however, skewed or delayed with respect to each other. This fact can, for example, be used to have three input streams share one input channel. An alternative use of the same property is to interleave the computations on three sets of equations. Correspondingly, the stacks storing intermediate values can either be reduced in length by pushing only every third value onto them, or reduced in number by combining stacks for three adjacent columns. If interleaving of different problems is used no reduction in the number or length of stacks can be made.

The flow of data in the array and the progression of the computations can be illustrated intuitively by stream lines for the data paths and wave fronts for the synchronization between the different data streams. Figure 14 shows the data streams in the elimination part and the forward substitution part of the array. The wave fronts can be drawn as in Figure 15. Wave fronts indicated by solid lines shows how the elimination factors propagates through the array. The dashed lines indicate how rows in the right-hand side of the equations enters the array.

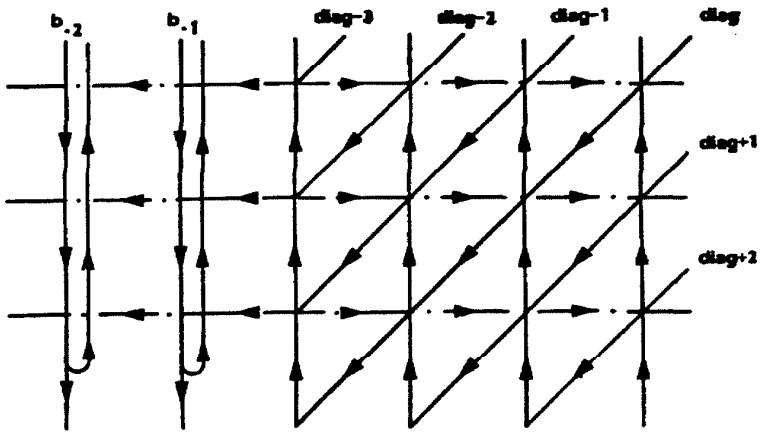


Figure 14. Stream lines

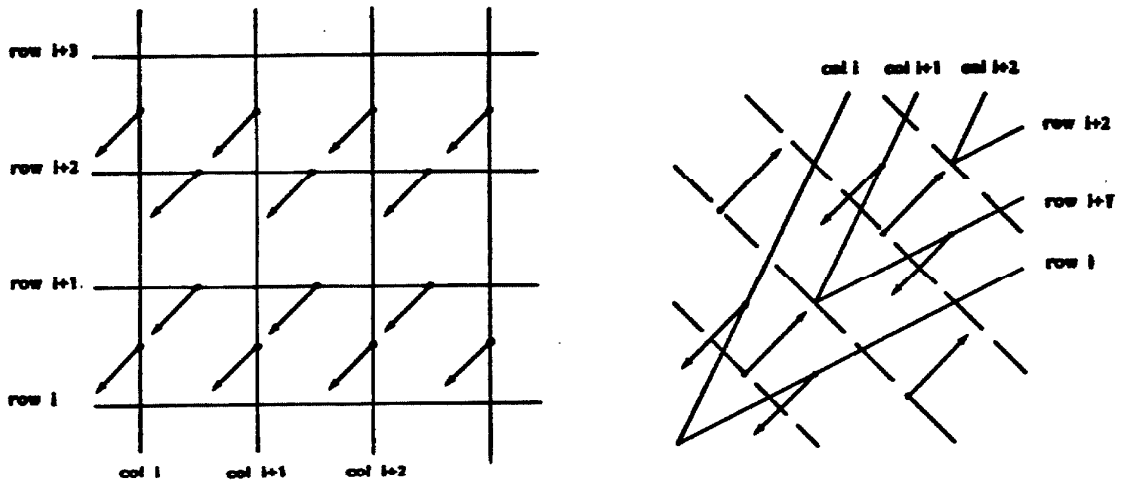


Figure 15. Wave fronts without and with full pipelining

3. Arrays performing partial pivoting.

3.1 Two-dimensional arrays.

Partial pivoting is in many cases sufficient to alleviate problems with numerical accuracy. Partial pivoting means pivoting on the largest element in a column. In the implementation shown in Figure 1 all rows are computed concurrently. The pivot element can be directly chosen by comparison. The computations can resume when the comparison has been made and a pivot element chosen. The data flow has to be adjusted according to the location of the pivot element and row. An implementation of an array performing partial pivoting is shown in Figure 16. The details of the pivot selection unit is left out. In addition to selecting a pivot row the selection unit also has to generate information for the other rows telling whether they are above or below the pivot row. This information is used to place the pivot row on the vertical wires and to route the other r rows to the proper rows of the array. The dashed lines in Figure 16 indicate how the pivoting information is distributed through the array. The input data is organized as in Figure 2. The contents of the stacks depend on the pivoting. It should be noticed that partial pivoting may increase the bandwidth of the matrix. The number of off-diagonals above the diagonal may be r+s instead of s.

The pivot selection mechanism can be extended to include a mechanism for the detection of singularity of the coefficient matrix. The product of the diagonal elements of the upper triangular matrix equals the product of the eigenvalues of the matrix.

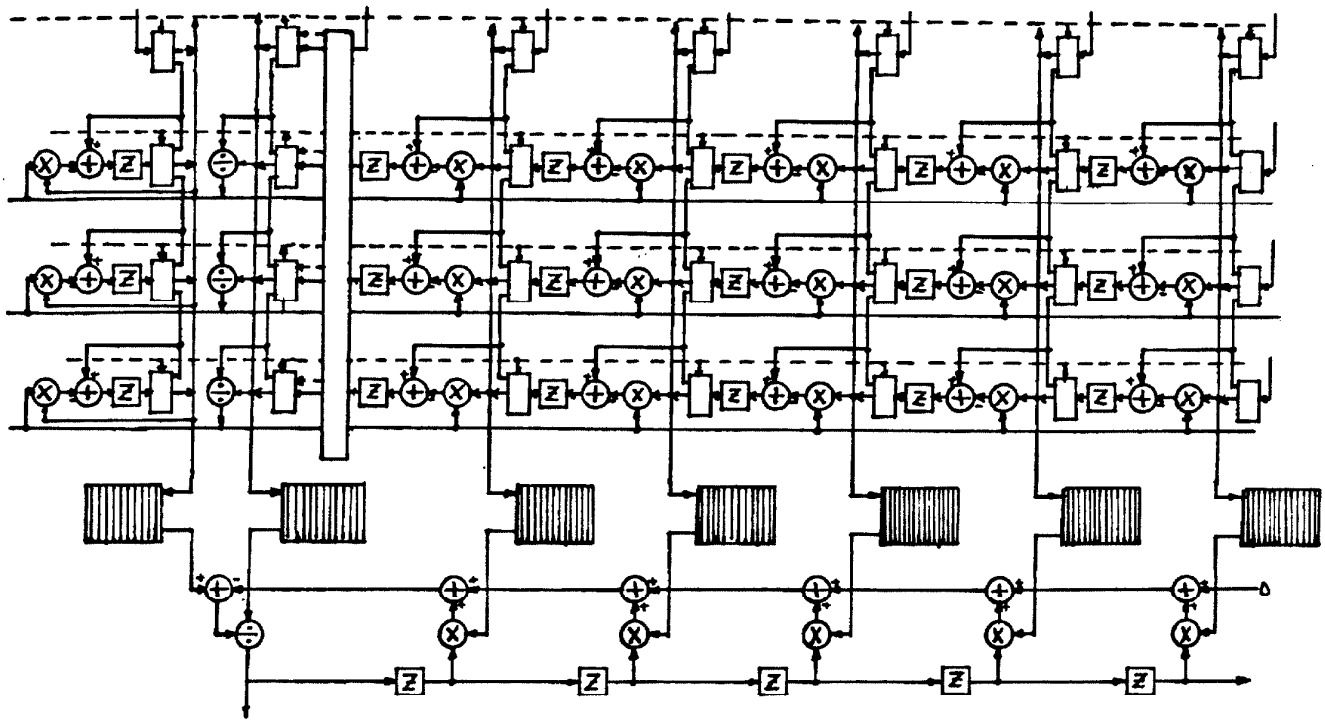


Figure 16. An array with partial pivoting

In the pipelined version according to Figure 11 pivoting is not accommodated as easily. The elements in a column are not available at the same time. They are computed successively with a computation starting on a new column every third cycle. This scheme cannot be preserved if pivoting is introduced. All computations in a matrix column have to be completed before the pivoting can be made. Implementing partial pivoting in the fully pipelined version of the array requires additional control compared to the nonpipelined version. Figure 17 shows an implementation.

The pivot selection mechanism of Figure 16 is required also in the implementation in Figure 17. However, the selection information is distributed through the array in a pipelined fashion. The basic 3-cycle that was implicit in the array in Figure 11 now has to be made explicit. Hence the shift register cells in the data paths either do not shift or shift during three consecutive cycles. The transition from a nonshifting state to a shifting one is controlled by the distance from the pivot element. The 3-cycle is initiated at the pivot element and propagates through the array. It progresses one row and one column per cycle. Pushing data onto the stacks is also controlled by the progression of the 3-cycle. The time required to perform elimination of one column including pivoting is in the range $[(r+1)/2]+3$ to $r+3$ plus the time required to select the pivot element. The selection of a pivot element and restarting the computations may take a time ranging from $\log_2 r$ to $2r$ cycles. By allowing a conection from the top to the bottom row of the array these estimates can essentially be reduced by a factor of two.

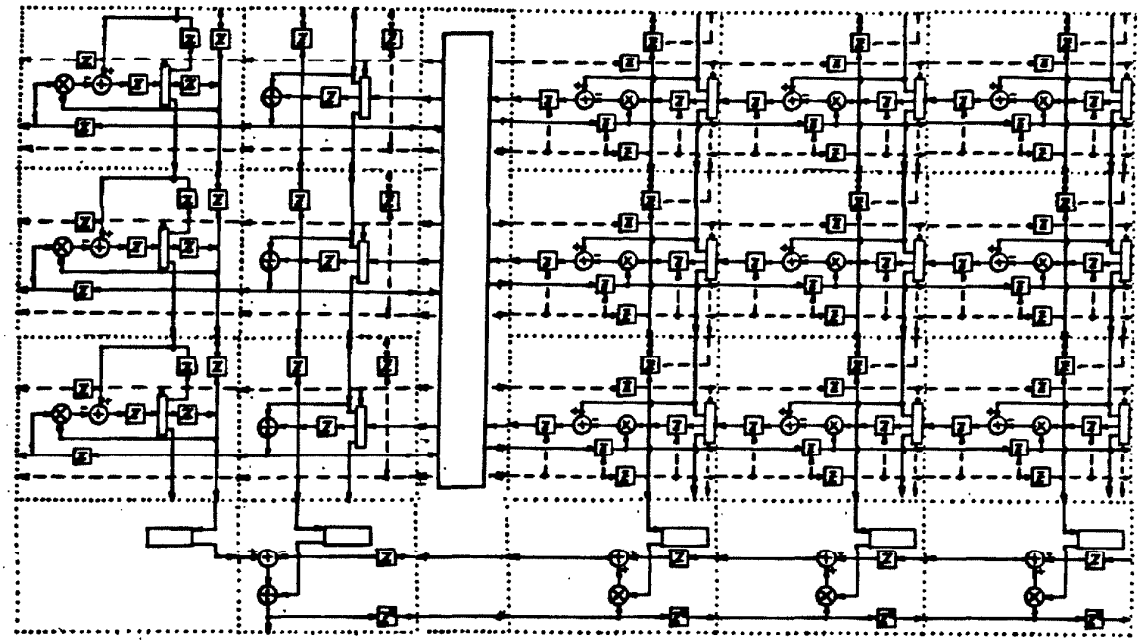
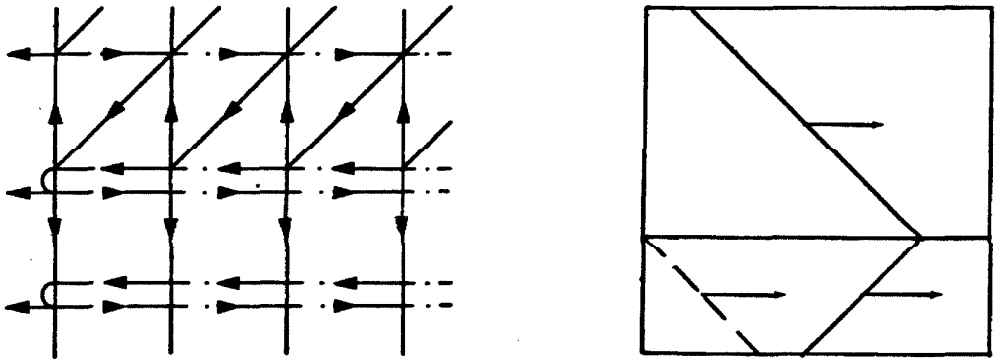


Figure 17. A fully pipelined array with pivoting.

Pivoting distorts the flow of data through the array. There is no longer a fixed route through the array. The stream lines and wave fronts are as in Figures 14 and 15 when pivoting is performed on the diagonal element. If pivoting is made on some other element in a column, a flow as in Figure 18 may occur. The solid lines in Figure 19 indicate the wave front corresponding to pivoting, as in Figure 18. The dashed line indicates the wave front that will propagate through the array if the next pivot element is the diagonal element.



Figures 18 and 19. Stream lines and wavefronts when pivoting on an off diagonal element

3.2 Linear Arrays.

In the two-dimensional arrays discussed above pipelining was introduced to avoid broadcasting. If partial pivoting is unnecessary pipelining represent no problem, but since the data flow is turbulent data has to be spaced out in time. If partial pivoting is necessary a penalty in performance has to be paid in that elements in a column has to be "lined up" before a new pivot row can be selected. Furthermore a few cycles (r/2 cycles on the average) may be required to restart the computations in the correct place. Hence, the utilization of the array is decreased.

It is possible to reduce the amount of hardware without loss of performance for Gaussian elimination with partial pivoting. A linear array is indeed sufficient to perform Gaussian elimination with partial pivoting in the same time as the two-dimensional array above does. Each module in the linear array will have r+s shift register cells. *Figure 20 shows the principle arrangement of the factorization and* forward substitution part.

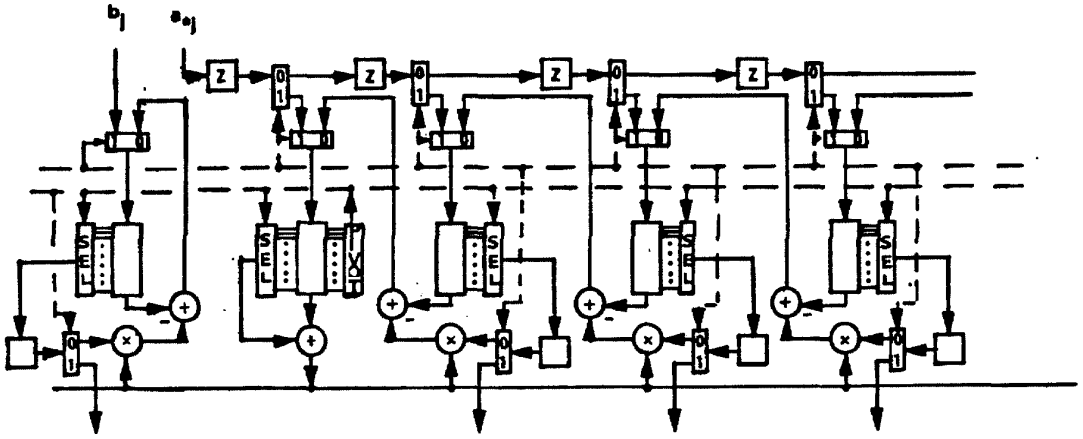


Figure 20. A linear array for Gaussian elimination with partial pivoting.

4. Arrays with fewer cells than what corresponds to the matrix bandwidth.

4.1 Arrays with storage of r*s matrix elements but fewer multipliers/adders.

In the previous sections it was assumed that the number of input channels was equal to the bandwidth of the matrix. If the bandwidth of the matrix is smaller than the number of input channels to the network the band can always be augmented with diagonals of zero value. The network will then produce the correct solution. The time to compute the solution will still be proportional to the size of the matrix.

If either r or s of a matrix, or both, are greater than the corresponding parameters of the array the straight forward approach above can no longer be used. The control of the arrays discussed so far is largely implicit in the interconnection pattern. It is

assumed that the diagonal of the matrix enters along the diagonal of the array. Gaussian elimination requires that the pivot row, i.e., s elements, multiplied by appropriate factors be subtracted from all r rows having a nonzero entry in the pivot column. It is not possible to first perform the elimination on a band around the diagonal and then treat off diagonal elements outside the band in subsequent passes. The elements in a row have to be computed before the row is used as a pivot row.

Denote r and s of the array with ra and sa and the corresponding parameters of the problem rp and sp.

rp=ra, sp>sa

Assume ra is greater or equal to rp but sp is greater than sa. In this case a portion of all rp rows can be treated concurrently in the array. To perform the computations on all elements in a set of rows the array can be designed to scan the band of the matrix. The array has to be redesigned to perform this scanning operation. Intermediate storage needs to be introduced as well as explicit control. We will carry out the discussion of modifications with reference to the implementation presented in Figure 1. The simple delay elements along the diagonals have to be replaced by a number of delay elements, more precisely [(sp-1)/sa]+1, where the [] denotes integer divide. Explicit control has to be introduced. The input data has to be organized into rp+sa+1 input streams for the matrix coefficients.

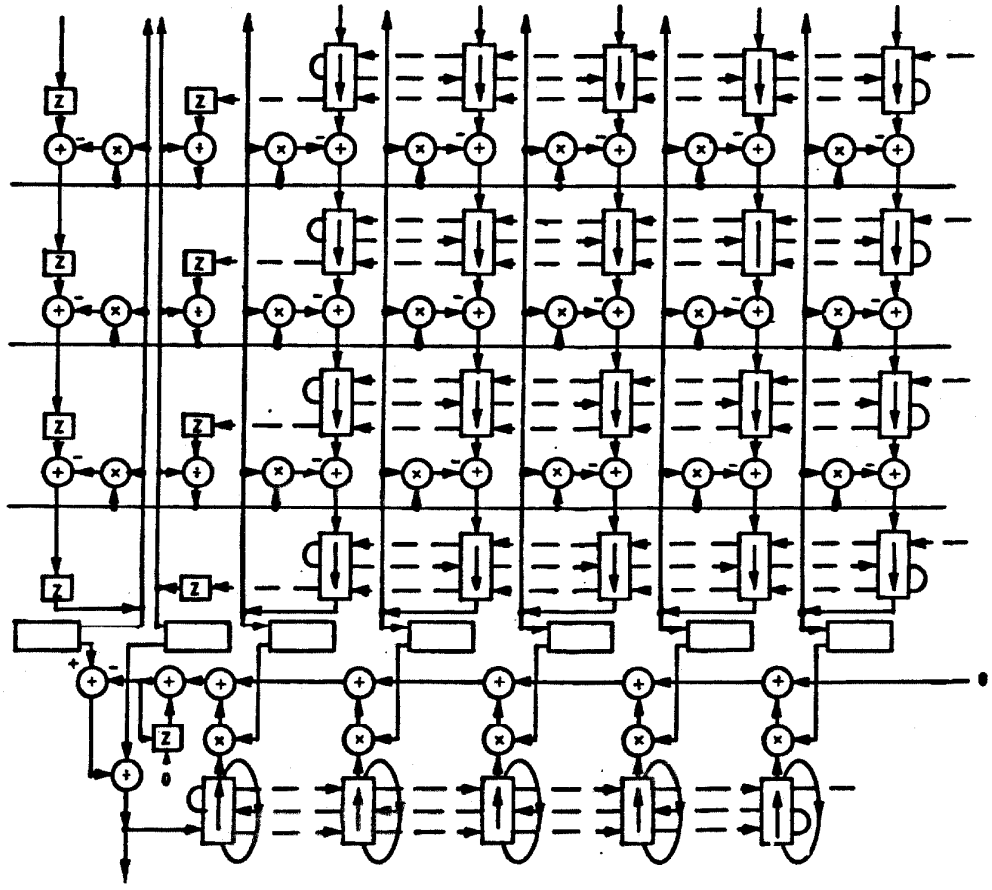


Figure 21. An array scanning a band from left to right.

Figure 21 shows an array scanning the band horizontally. All computations in a frame, Figure 22, are performed concurrently. The frames are treated from left to right in the horizontal direction. When all frames in the horizontal direction are processed the set of frames treated starts one row below and one column to the right of the previous set of frames. Figure 23 gives the data organization.

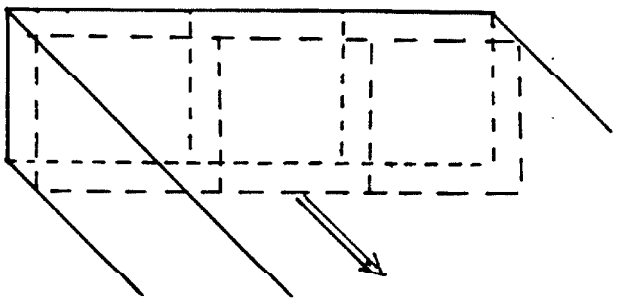


Figure 22. Computational windows.

$$
\begin{array}{llllll}
 & a_{512} & a_{513} & a_{514} & a_{515} & a_{516} \\
 & a_{511} & a_{510} & a_{59} & a_{58} & a_{57} \\
 & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 & a_{412} & a_{413} & a_{414} & a_{415} & a_{416} \leftarrow a_{517} \\
 & a_{411} & a_{410} & a_{49} & a_{48} & a_{47} \\
a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46}
\end{array}
$$

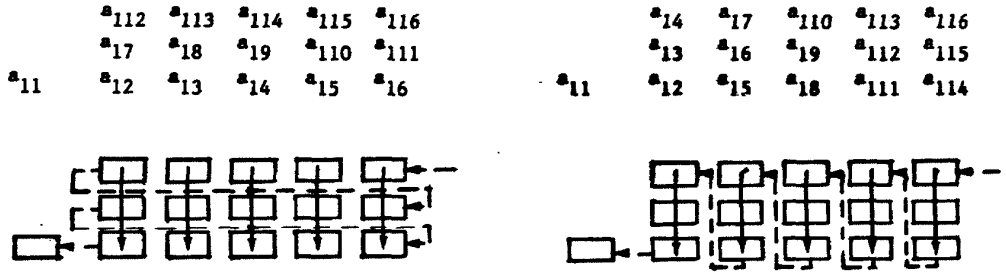
$$
\begin{array}{llllll}
 & a_{312} & a_{313} & a_{314} & a_{315} & a_{316} \leftarrow a_{417} \\
 & a_{311} & a_{310} & a_{39} & a_{38} & a_{37} \\
a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36}
\end{array}
$$

$$
\begin{array}{llllll}
 & a_{212} & a_{213} & a_{214} & a_{215} & a_{216} \leftarrow a_{317} \\
 & a_{211} & a_{210} & a_{29} & a_{28} & a_{27} \\
a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26}
\end{array}
$$

$$
\begin{array}{llllll}
 & a_{112} & a_{113} & a_{114} & a_{115} & a_{116} \leftarrow a_{217} \\
 & a_{111} & a_{110} & a_{19} & a_{18} & a_{17} \\
a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16}
\end{array}
$$

Figure 23. Data organization used in Figure 21.

An array according to Figure 21 stores rp*sp matrix elements but has only ra*sa multiply add cells. The area savings compared to an rp*sp array is hence due to a fewer number of multiply/add cells. The array of Figure 21 has the same basic data flow as the array in Figure 1. Data in ra rows of a matrix column are treated concurrently by one column of the array. The order in which the matrix columns are treated by the array is a design issue. As long as all ra elements in a matrix column are treated concurrently in a column of the array a correct result will be obtained. The order in which the matrix columns are treated by the array does, however, affect the control and the internal communication of the array. The data organization of Figure 23 implies local communication only. Alternative data organizations and their corresponding communication requirements are shown in Figures 24 and 25. A data organization according to Figure 24 requires communication paths of a length corresponding to sa multiply/add cells. In Figure 25 there are paths of length sp/sa shift register cells.



Figures 24 and 25. Alternative data organizations and associated communication requirements.

Figures 23 - 25 illustrate the affect the data organization has within rows on the communication. In Figure 24 an array column treats matrix columns at a distance sa consecutively. In Figure 25 the matrix columns treated concurrently by the array comes out of sa frames. Each array column treats sp/sa consecutive matrix columns.

The data organizations in Figures 23-25 are all based on the assumption that the overall data flow is diagonal and that pivoting is made on the diagonal elements, as in Figure 1. The data flow is designed so that successive diagonal elements appear successively in the same position of the array. This property simplifies the control and reduces the communication requirements.

The control of the array in Figure 21 is not as simple as in Figure 1. The multiply/add cells active in the first phase of the elimination process operates at a different frequency than division units and the forward substitution part. An entire row of the matrix and each component (row) of the right-hand side of the equation shall be multiplied by the same factor irrespective of how many partitions are used. Hence each multiply/add cell shall perform sp/sa operations using the same factor for multiplication. In the array in Figure 21 a vertical shift in the matrix part is made

every cycle while a horizontal shift takes place every sp/sa cycle. A shift in the part corresponding to the right hand side of the equation also takes place every sp/sa cycle. Correspondingly, sp/sa cycles are required in the back substitution part for the computation of every new component of the output vector. An accumulation of partial sums has to be performed during sp/sa cycles as indicated in Figure 21. The accumulator is reset when a new value of x is computed and an old value shifted out of the array holding sp components of x.

Broadcasting can be avoided in the implementation of Figure 21 in the same way as was discussed earlier. Introducing a delay between adjacent cells in the horizontal direction implies delaying the computations of consecutive columns one step with respect to each other. The horizontal shift in the shift register arrays also has to be pipelined and intermediate storage cells introduced for shifts to the right. There is no right shift using the data organization in Figure 25. Introducing delays in the vertical direction to avoid broadcasting will cost another cycle per row, if the communication topology is not changed and the control is not made more complex. Partial pivoting can be introduced into the array of Figure 21 in much the same way as in Figure 16. Partial pivoting can also be introduced into a pipelined version of the array in Figure 21.

rp>ra, sp=sa

In this case the scanning of the band has to be made in the vertical direction. The array is capable of treating ra rows concurrently. The array can be designed to treat ra consecutive rows concurrently, i.e., a frame of ra*sa matrix elements, or ra rows from ra different frames. In the latter case one array row can be made to successively treat rp/ra consecutive rows of the matrix. The computations can be performed in many other ways too. Figure 26 shows the basic computational behavior of an array scaning the band in a vertical direction. Figure 27 gives an array in which the matrix rows treated by consecutive array rows are located a distance ra apart. The array stores rp*sp matrix elements. The number of multiply add cells is ra*sa. The data organization is shown in Figure 28.

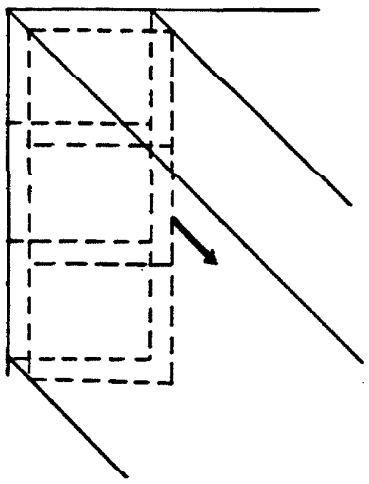


Figure 26. Computational windows for the array in Figure 27.

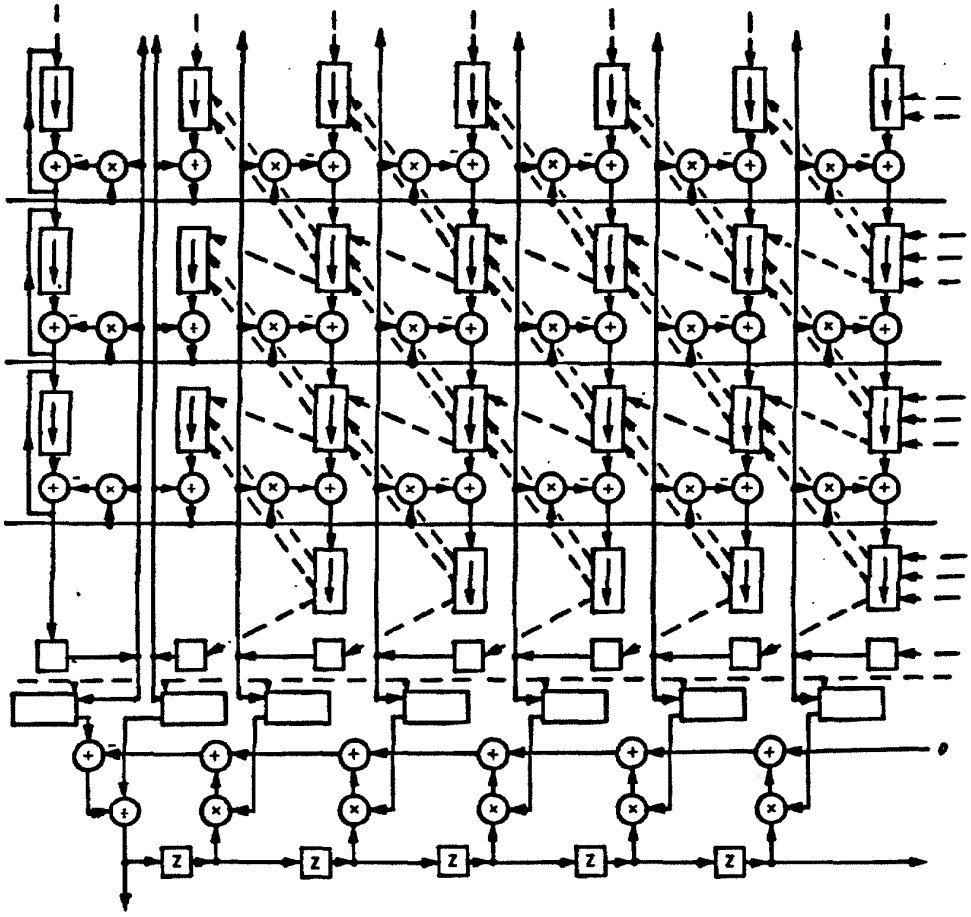


Figure 27. An array scanning the band vertically.

$$
\begin{array}{llllllllll}
 & a_{123} & a_{124} & a_{125} & a_{126} & a_{127} & a_{128} & & & \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & & \\
 & a_{112} & a_{113} & a_{114} & a_{115} & a_{116} & a_{117} & & & \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & & \\
b_{10} & a_{101} & a_{102} & a_{103} & a_{104} & a_{105} & a_{106} & & & \\
b_9 & a_{91} & a_{92} & a_{93} & a_{94} & a_{95} & a_{96} & \leftarrow a_{107} & \leftarrow & a_{118} \\
b_8 & a_{81} & a_{82} & a_{83} & a_{84} & a_{85} & a_{86} & \leftarrow a_{97} & \leftarrow & a_{108} \\
b_7 & a_{71} & a_{72} & a_{73} & a_{74} & a_{75} & a_{76} & \leftarrow a_{87} & \leftarrow & a_{98} \\
b_6 & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & \leftarrow a_{77} & \leftarrow & a_{88} \\
b_5 & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & \leftarrow a_{67} & \leftarrow & a_{78} \\
b_4 & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & \leftarrow a_{57} & \leftarrow & a_{68} \\
b_3 & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & \leftarrow a_{47} & \leftarrow & a_{58} \\
b_2 & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & \leftarrow a_{37} & \leftarrow & a_{48} \\
b_1 & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & \leftarrow a_{27} & \leftarrow & a_{38}
\end{array}
$$

Figure 28. Data organization for an array scanning the band vertically.

The data flow in the array in Figure 27 is basically the same as in Figure 1. A matrix column is treated by an array column. The flow of data is from the upper right hand corner to the lower left hand corner. The data is organized so that consecutive diagonal elements appear successively in the same location. No explicit control and corresponding additional communication is necessary for pivoting on diagonal elements. Figure 29 shows a different data organization and the corresponding communication requirements. The data organization corresponds to treating ra consecutive rows concurrently in the array. Consecutive diagonal elements appear successively in the same position, as in Figure 28. As is apparent from Figure 29 long communication paths are required with this alternative data organization.

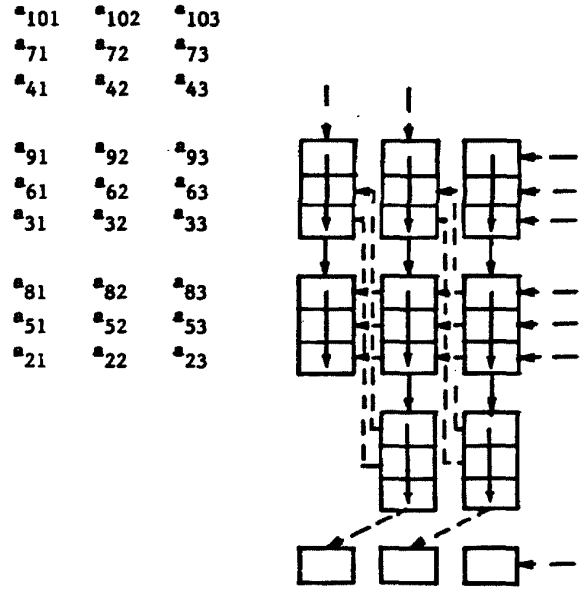


Figure 29. An alternative data organization and associated communication paths.

The pivoting row has to stay on the vertical wires for all rp/ra cycles required to carry out the vertical scanning. The components of the right-hand side of the equations have to be scanned also, as is indicated in Figure 27. The stacks are controlled in the same way as are the vertical wires. At every rp/ra cycle a shift in the direction of the dashed arrows should be performed. At the same time new elements are pushed onto the stacks. The back substitution part is the same as in Figure 1 since the number of elements above the diagonal is assumed to be sa.

Broadcasting in the horizontal and vertical direction can be avoided by introducing delays between adjacent multiply/add cells. Introducing delays in the horizontal direction costs a cycle for each rp/ra cycle. Delays in the vertical direction cost another cycle. No change in the communication paths of the array is required. The control should be pipelined in the same way as the data flow.

Partial pivoting can be introduced in the array of Figure 27 the same way as in Figure 16 and in a way similar to Figure 17 for a pipelined version of Figure 27.

ra<rp, sa<sp

In the case where ra as well as sa is smaller than the corresponding values of the problem, rp and sp respectively, scanning has to be performed both horizontally and vertically. The number of storage cells required in the nonpipelined version is still rp*sp. The storage cells are distributed evenly among the ra*sa multiply/add cells used in the elimination. For each right/hand side ra storage cells are required in the forward substitution part and sa storage cells required for components of the solution vector in the back substitution part of the array.

The control of the array becomes more complex. Several alternatives exist. One alternative is as follows. The scanning in the horizontal direction is performed first. The factors distributed horizontally are kept constant during sp/sa cycles. A new set of factors are then computed for a new set of ra rows. The computations on these rows are performed during the next sp/sa cycles. After rp/ra such cycles, the elimination of one column is complete, and so is the forward substitution. Hence one of the inputs to the dividers, the pivot element, is constant during the sp/sa*rp/ra cycles required to eliminate a column. The other input changes every sp/sa cycles and rp/ra storage cells are required to store the input to one divider.

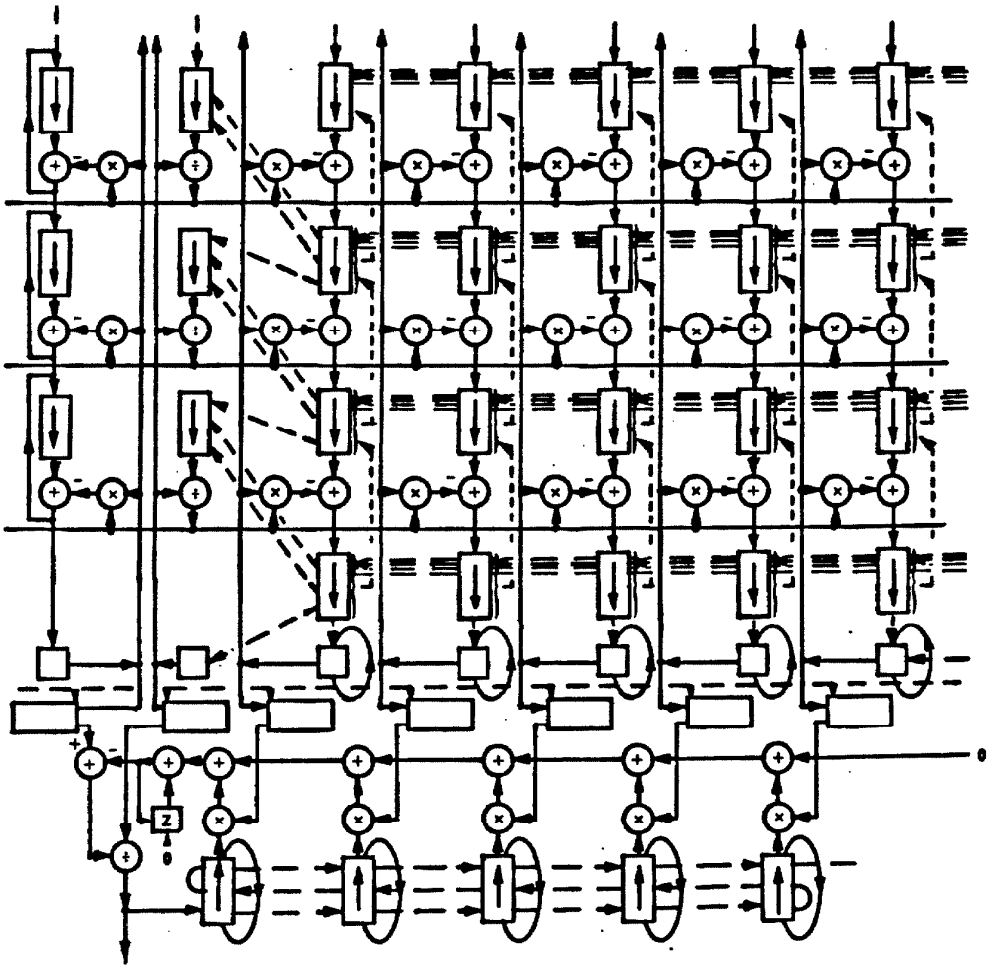


Figure 30. An array scanning the band horizontally and vertically.

After the elimination of one column a horizontal shift as in Figure 21 brings a new column into the array. This column should be immediately to the right of the rightmost column in the array. The elements shifted out of the leftmost column of the multiply add cells goes to the dividers as indicated in Figure 27. Instead of having one storage cell associated with each vertical line in the multiply/add cells we now have sp/sa cells for the elements of the pivot row. Shifting the information stored in the columns down by sp/sa cells moves the previous pivot row into the stacks and the row to be used next into the correct position for the elimination of the next column. However, before the elimination can proceed it is also necessary to copy or shift the content of the storage of each multiply/add cell to the storage of the multiply/add cell immediately above in the same column. The elimination of elements in the next column can now start. During the elimination of a column the elements of the pivot row are recirculated rp/ra times. Figure 30 illustrates an array scanning the band both vertically and horizontally.

Block oriented algorithms and arrays.

The algorithms used in alternative 1 to handle problems of dimensions larger than the size of the array resulted in shift register lengths that were problem dependent. It may be preferable to implement the variable length shift registers as a random access memory. The simplicity of the control of the array is lost.

We will now propose another array where the array is independent of the problem size. Storage of intermediate results is accomplished outside of the array. The control is the major source of difference between the array now being proposed and the basic arrays described earlier. The algorithm is based on block operations. Each block is assumed to be of a size that fits within the array. We assume there are Sb nonzero blocks above the diagonal and Rb nonzero blocks below the diagonal. One way to perform the block operations is to start with the first diagonal block and perform the factorization and forward substitution as outlined in the first part. Storing the columns of the lower triangular part as they are computed, they can be used to compute the first row of blocks of U. Storing the blocks of U, the block rows with nonzero blocks in the column of the pivot block can be properly computed. Below we divide these computations into two steps, one corresponding to the computations of the blocks in the pivot column, one to the remaining blocks. The computations can be described by the following equations.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 & 0 \\ A_{21} & A_{22} & A_{23} & A_{24} & 0 & 0 \\ 0 & A_{32} & A_{33} & A_{34} & A_{35} & 0 \\ 0 & 0 & A_{43} & A_{44} & A_{45} & A_{46} \\ 0 & 0 & 0 & A_{54} & A_{55} & A_{56} \\ 0 & 0 & 0 & 0 & A_{65} & A_{66} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix}$$

$$\underline{\text{Step 1}} \quad L_{11}U_{11} = A_{11} \; ; \quad y'_1 = L_{11}^{-1}y_1$$

$$\underline{\text{Step 2}} \quad L_{11}^{-1}A_{12} = U_{12} \; ; \quad L_{11}^{-1}A_{13} = U_{13}$$

$$\underline{\text{Step 3}} \quad L_{21}U_{11} = -A_{21} \; ; \quad y'_2 = y_2 - L_{21}y'_1$$

$$\underline{\text{Step 4}} \quad A'_{22} = A_{22} - L_{21}U_{12} \; ;$$

$$A'_{23} = A_{23} - L_{21}U_{13} \; ;$$

The flow of the data through the array varies during the different steps. The matrix coefficients flows along the diagonals as in the basic array during the first step, assuming no pivoting. Figure 31 attempts to illustrate the data flow, organization and external control during the four steps and the back substitution step.

During step 2 when the blocks in the pivot row are treated the flow is vertical. The matrix elements enters at the top and the columns of the lower triangular diagonal factor enters from the left and moves to the right. The matrix blocks can be supplied in a pipelined manner. Pipelining is possible due to the triangular shape of the diagonal block of L. Even though a matrix entering the array meets the rows from the previous block moving in the opposite direction the rows of the new block will not be changed. The rows of the old block will be multiplied by zero. When a row reaches the bottom of the array it is shifted to the vertical wires and fed back into the array. Also, all computations required to make a row of A into a row of U has been made. The lower triangular factor is cycled and entered as many times as there are nonzero blocks above the diagonal in the pivot row. The dividers are not used in this part of the computation.

It should be noticed that the computations performed on the blocks of the matrix that falls in the upper right half are the same computations that need to be performed on the right hand side of the system of equations. Hence, if there are more equations to be solved for the same matrix than can be handled by the part of the array that does the forward elimination, the factorization part can be used as in step 2.

In step 3 the nonzero blocks in the pivot column below the diagonal are eliminated. The blocks are entered from the right, one column per cycle. We assume that the blocks are entered in row order. The computations of the blocks in the lower triangular matrix can be pipelined, as in step 2. The columns of the new block will not be changed until the first of its columns reaches the left side of the factorization part of the array. This property is due to the fact that the diagonal block of U is triangular. The columns of the new block do not change since the values of U supplied to the multipliers are zero as the block moves from right to left. The upper triangular matrix corresponding to the pivot block is cycled through and applied to the array as pivot rows as many times as there are nonzero blocks below the diagonal in the pivot column. Hence, the rows of the upper triangular matrix move upwards from the bottom. The blocks in the pivot column of the lower triangular matrix appear at the left side of the factorization part of the array in the same order as the column blocks are supplied to the right side.

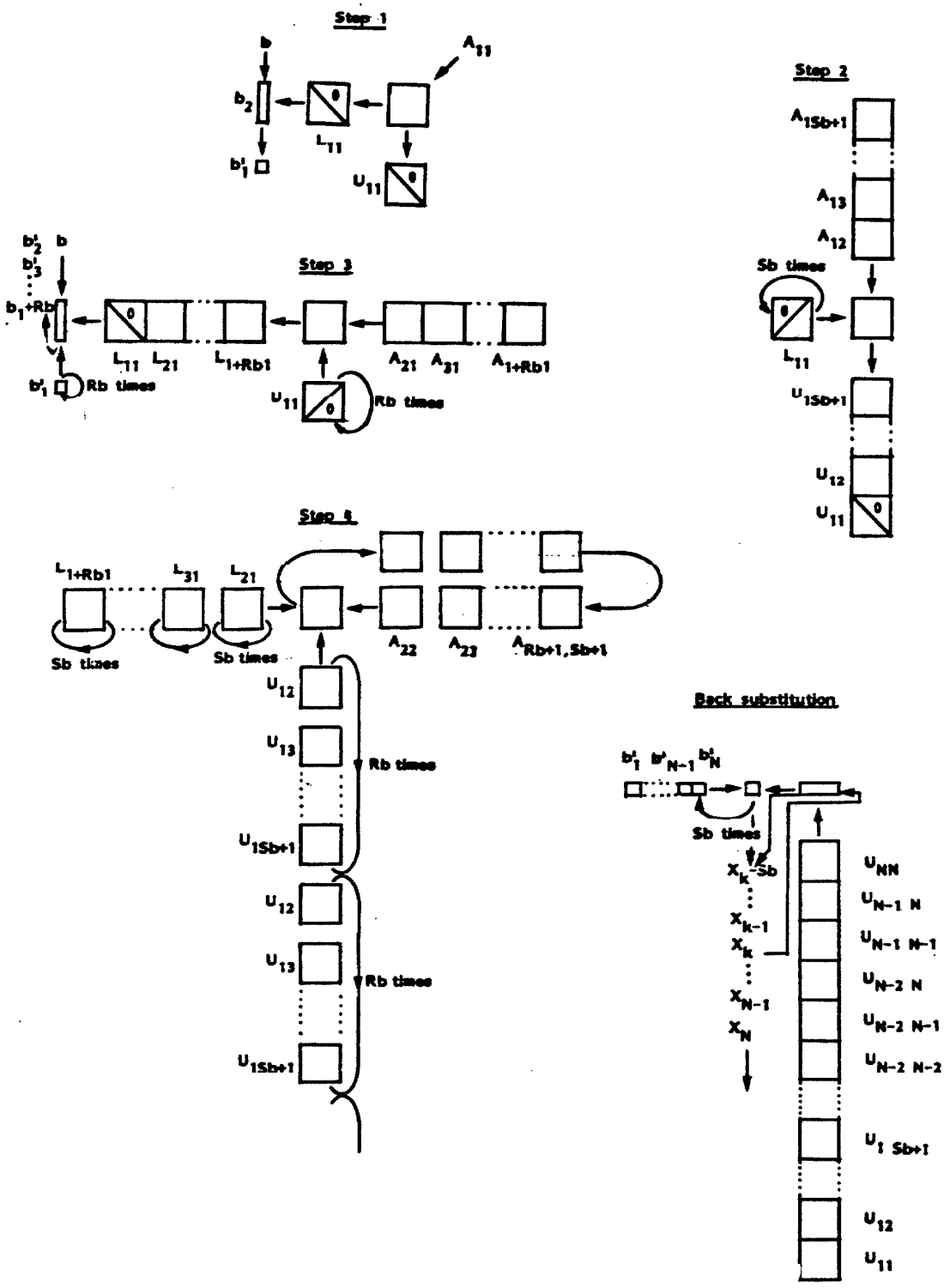


Figure 31. Data flow and organization in an array for block operations.

It should be noticed that there is a great deal of symmetry between step 2 and step 3. The data flow in the array for one of these steps is essentially the flow of the other step mirrored in the diagonal or transposed if expressed in a mathematical framework. The computations are not identical, however. The symmetry can be used to reduce internal wiring of the array. The external data organization become slightly more complex if the symmetry is used advantageously. The symmetry is not exploited in Figure 31. Figure 32 shows the data organization in step 2 when the symmetry is exploited.

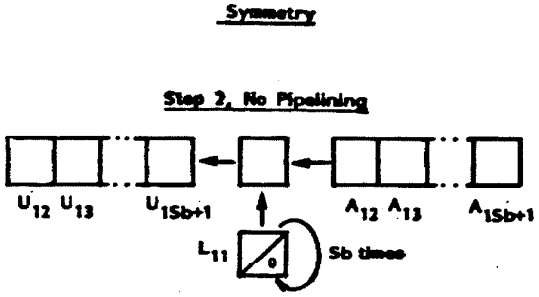


Figure 32. Data organization in step 2 exploiting symmetry of block operations.

In steps 1 through 3 the pivot block was factorized and the forward solution on the first set of components of the right member computed. The second step computed the blocks in the pivot row of the upper triangular matrix. The third step computed the blocks in the pivot column of the lower triangular matrix and performed the forward substitution on sets of components of the right member. What remains to be done before a new pivot element can be selected and the computations proceed on a matrix with one block row and column less, is to perform the proper updates on the rb*sb blocks immediatedly below the pivot row and immediately to the right of the pivot column. The computations can proceed in several different ways, only affecting the storage management outside the array. If the size of the blocks is n by n then a total of n^3 operations are to be made in the forth step. Since there are n^2 cells in the array at least n steps are required for the computation. A loop around the adder can be introduced for the accumulation of products. The updated blocks will be part of the input stream when the elimination starts on the block matrix of dimension one less than the one just being treated.

In Figure 31 it is assumed that the computations in step 4 proceed row by row. With this order of computation each of the lower triangular matrices should be applied Sb times. The upper triangular blocks in a row is supplied in sequence and the whole row applied Rb times. It is also possible to perform the computations interchangeably row and columnwise. The pattern in which the blocks of the upper triangular and lower triangular matrix are applied has to change accordingly. When the computations proceed rowwise the blocks in the row are supplied sequentially and the lower triangular block corresponding to that row applied as many times as there are nonzero blocks in the pivot row. When the computations proceed columnwise the roles are interchanged, that is, the blocks in a column are supplied sequentially and the block in the upper triangular part that corresponds to the column being treated applied as many times as there are nonzero blocks in the column.

Hence, the output of the factorization part of the array is of three kinds; blocks belonging to the lower left triangular factor, appearing on the left side of the array, blocks belonging to the upper right triangular factor appearing at the bottom of the array, and updated matrix blocks belonging to the part of the matrix that yet has to be factored.

In the back substitution part there are two control frequencies. There is one required for the back substitution on the off diagonal non-zero blocks. During this phase the right-hand side is updated Sb times, one for each off diagonal block. During this phase the values of Sb partitions of x are supplied to the array. The right-hand side is updated Sb times only after the first Sb partitions of x are computed, which should be obvious. When all off-diagonal blocks in a row are treated a new partition of x is computed. A diagonal block of the upper triangular matrix is supplied to the array for back substitution. The partitions of the right-hand side are advanced one step as is the array for the solution vector x.

Pipelining can be introduced in the same way as in the preceding cases without any particular problem. Step 1 is identical to the basic case discussed before. Pipelining to eliminate broadcasting in steps 2 and 3 requires the data to be spaced at a distance two. In step four the accumulation of products is stationary. Data need not be spaced out as in the other three steps. Assume that the array is symmetric, i.e., rp = sp = n. Then the first step will require n + 3n cycles. For each block 2n cycles are required for step 2. For step 3 n cycles are required to load the first block if the symmetry is not exploited, otherwise the loading can be pipelined with the final computations of step 2. Each block in step 3 requires 2n cycles. The loading of the first block in step 4 can be pipelined with the last set of computations of step 3. For each additional block n cycles are required for the loading. Each block requires n cycles for the computation to be completed. The number of cycles to perform all computations related to the elimination of a block column is as follows.

	One column	Total
Step 1	3n	n + 3nN
Step 2	2n*Sb	2n(N-(Sb+1)/2)Sb
Step 3	2n*Rb + (n)	2n(N-(Rb+1)/2)Rb + (n(N-1))
Step 4	2n*Rb*Sb	2n(N-max(Sb,Rb))*Sb*Rb + + 2n*min(Sb,Rb)(max(Sb,Rb) - - min(Sb,Rb))(max(Sb,Rb) + + min(Sb,Rb) - 1)/2 + + 2n*min(Sb,Rb)(min(Sb,Rb) + 1)(2min(Sb,Rb) + 1)/6

Disregarding the "edge" effects, the total time is O(nN(Sb*Rb+Sb+Rb)). The size of the matrix is nN by nN and its bandwidth n(Sb+Rb+1). The time required by a sequential machine is $O(n^3N*Sb*Rb)$. Hence, the reduction in time is $O(n^2)$, as should be expected.

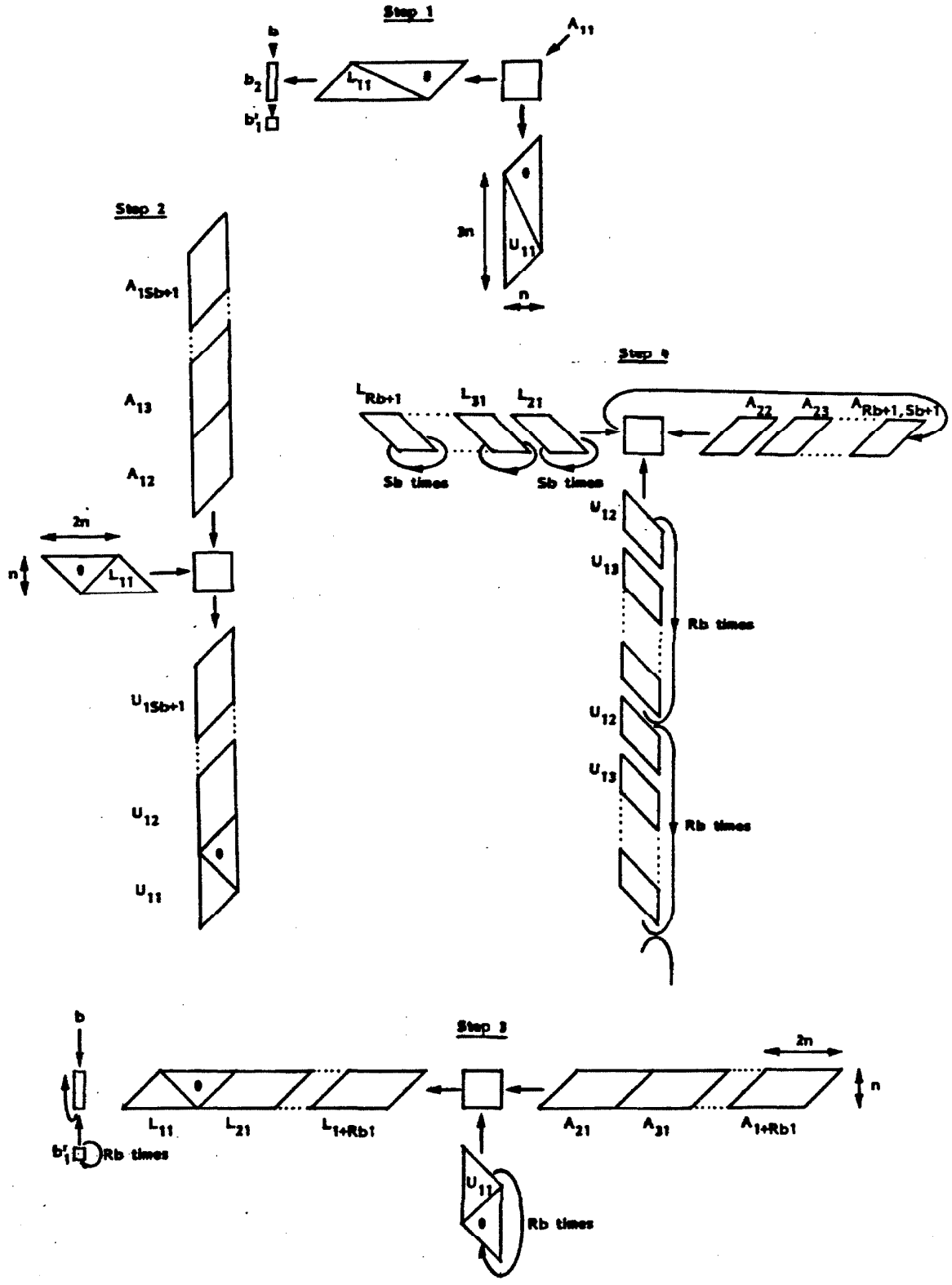


Figure 33. Data flow and organization in a fully pipelined array for block operations.

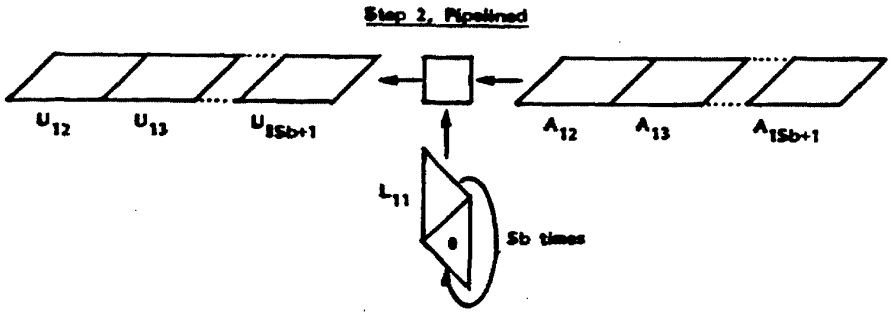


Figure 34. Data organization in step 2 exploiting symmetry.

Pivoting is not as easy to introduce as in the basic case since only a fraction of the matrix resides in the array at any given time. Pivoting within the diagonal block that is supplied to the array in step 1 can be accomplished in the same way as in the basic array. It is, however, necessary to save the information that identifies the rows that are used as pivot rows. The control becomes more complex, as described in the discussion of the pipelined array that includes features for pivoting.

A few cells for arrays performing the computations according to steps 1-4 exploiting symmetry is shown in Figures 35-37. Figure 38 shows where the cells fit in an array. In Figure 17 a pipelined array capable of performing pivoting was shown. The alternative discussed here requires that the cells be somewhat changed. Figure 34 shows a cell in the factorization part of the array. Two bits are required for the selection of data route, as in Figure 17. There are, however, four alternative paths in Figure 35 instead of three as in Figure 17. Path A is selected during step 1 when the pivot row is located below the current row in the array (above in the matrix). Path B is selected when the current row is the pivot row and path C when the pivot row is located above the current row in the array (below in the matrix). Path C is also chosen as the path for the data in steps 2 and 3. It is also used during the loading and unloading phase of step 4. Path D is used during the computing phase of step 4. The control lines are dashed. There are two kinds of control lines. One kind carries the selection information just discussed. The other carries the shift information for the shift register cells. Delay elements with no explicit control wire, delays the information one cycle. The partitioning of the array into cells can be made in several ways. The one chosen in Figure 35 simplifies and reduces the number of edge cells.

Figure 36 shows a cell for the pivoting column of the array. The difference compared to Figure 17 is that the output of the divider can be separated from the horizontal wires for the data flow. This is used in step 4 when the dividers are not used and blocks of L are supplied to the cells in Figure 35.

The forward substitution on the right hand side of the equations can be performed by cells according to Figure 37. The cells are functionally identical to those in Figure 17 but drawn slightly different. The selector circuit has been split into two and a shift register cell (word) been saved.

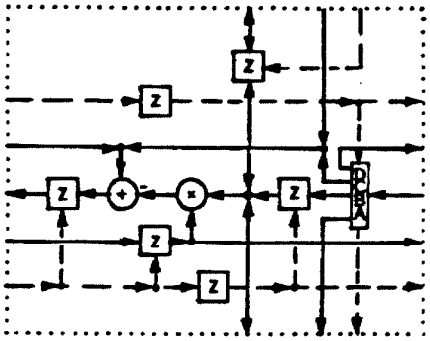


Figure 35. A typical cell in the elimination part of the array.

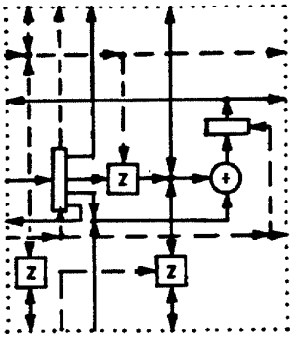


Figure 36. A typical cell in the pivot column of the array.

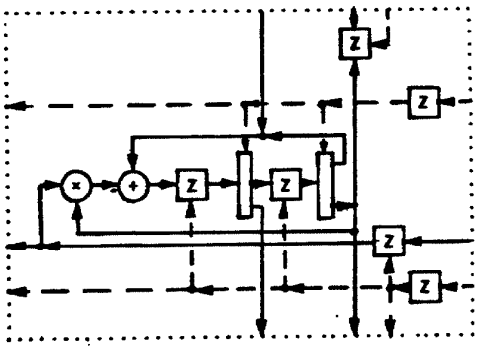


Figure 37. A typical cell in the forward substitution part of the array.

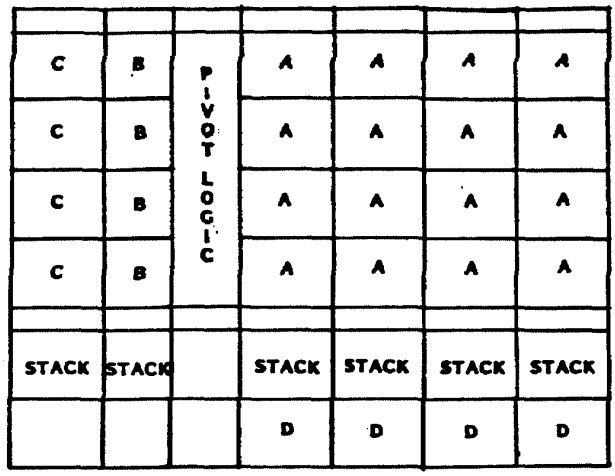


Figure 38. Placement of the cells in Figures 35 - 37 in an array.

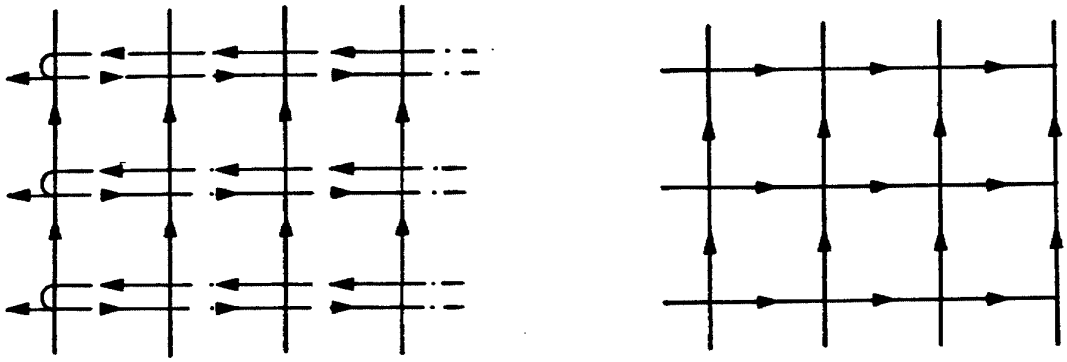
With n by n cells of the type shown in Figure 35 and a column of n pivot cells, Figure 36, the factorization of an n+1 by n+1 matrix can be performed. In step 2 of the computations the diagonal of a diagonal block of the lower triangular matrix is supplied to the pivot column. The diagonal is not stored but is 1. During step 2 a block of the pivot block- row of the matrix is supplied to the array, one column per row of the array. The diagonals below the main diagonal of the diagonal block of L are supplied to the columns of the array. Hence it would be possible to treat a block with n columns and n+1 rows in the array in step 2. In step 3 a row in a block of the matrix is supplied to a row in the array. The main diagonal of the diagonal block of U is supplied to the pivot column. The diagonals above the main diagonal are supplied to the columns of the array in order. Hence in step 3 blocks of n rows and n+1 columns can be treated. Finally in step 4 blocks of n rows and n columns can be treated. In the interest of keeping the block size constant through all steps the top row and rightmost column of the array should not be used during step 1 and the rightmost column not in steps 2 and 3. This can be accomplished by supplying the top cell in the pivot column by a zero in step 1 and the bottom cell in the rightmost column by a zero in steps 1-3.

To speed up the loading and unloading during step 4 a bypass of the adders can be included at the expense of additional control. Otherwise, 0 should be supplied to the columns of the array, except for the pivot column that should get a 1.

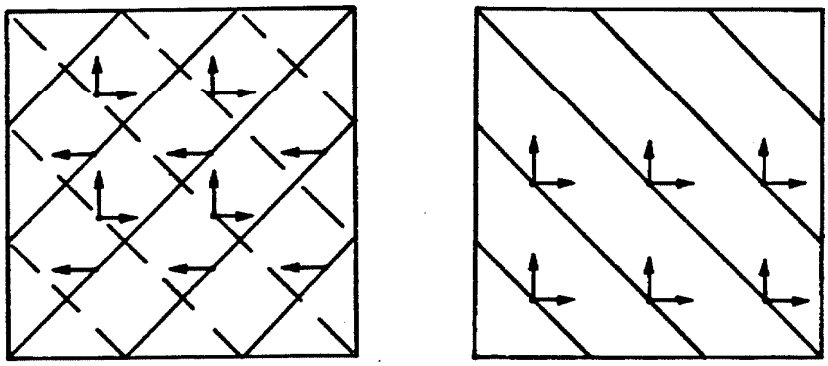
The control is different during the various steps. If the pivoting always is made on the diagonal elements then the data can be shifted as in Figure 11 for step 1. With pivoting the pivot selection circuit selects the pivot element and the computations resume from its location. The information on the shift wires propagates out through the array from the place of the pivot element. The shift information consists of a request to shift during three consecutive cycles followed by a no-shift request until

further notice. In steps 2 and 3 the shifting can be made continuously. However, the data has to be spaced at a distance two. In step 4 the shifting both during the loading/unloading phase as well as the computation phase should be made continuously.

The stream lines and wave fronts in step 1 are identical to what was shown in Figures 18 and 19. The stream lines in steps 2 - 4 are shown in Figures 39 and 40. The wave fronts are indicated in Figures 41 and 42.



Figures 39 and 40. Stream lines in steps 2 and 3, and step 4 respectively.



Figures 41 and 42. Wave fronts in steps 2 and 3, and 4 respectively.

Summary

In the array structures proposed above the model of storage elements as proposed in Cohen [1] has been used. A formal derivation and verification of the arrays can be made using this notation, Johnsson et al., [2], but is not carried out here.

Stream lines are an abstraction of the flow of data through the array that helps in the understanding of the behavior of the arrays. Wave fronts are another related abstraction of the behavior of the arrays that focuses on the synchronization between different data streams.

In the arrays proposed the stacks are used both during the elimination and forward substitution phase as well as the back substitution phase. All other parts are only used in one phase. By providing a double set of stacks the solution of a set of problems can be pipelined. The stack lengths can be reduced by pushing values only every third cycle onto them in the fully pipelined alternative of the arrays. Alternatively, the number of stacks can be reduced by multiplexing the output from three adjacent columns into one stack.

Introducing delays between adjacent cells in the horizontal direction implies delaying input streams for data above the diagonal by one step for each diagonal away from the main diagonal. Correspondingly, introducing delays in the vertical direction implies delaying the input streams corresponding to diagonals below the main diagonal by one cycle for each stream away from the main diagonal. The total delay between adjacent streams corresponding to diagonals below the main diagonal will hence be two since there is already a need to delay the streams by one cycle due to the delays on the diagonals of the array.

The fully pipelined version of the array has an operational behavior such that a cell is active during three consecutive cycles and then becomes idle until a new request for computations is transmitted to it. The control of the data flow is distributed throughout the array. The control is built into the interconnection scheme in the naive array. In the pipelined array that includes pivoting the control is made explicit. The control signals are, disregarding the loading phase, initiated from the pivoting logic and properly forwarded from one cell to the other. There are no global communication paths. The control information is also pipelined.

A multiply add cell has three inputs. One input corresponds to a diagonal of the matrix, one to another diagonal, and one to multiplicands used to eliminate column entries falling on a diagonal. The diagonals a cell receives are at a distance j apart, where j denotes rows in the array numbered from the bottom. Hence, j=1 for the bottom row and ra for the top row. The data flow is designed so that when a diagonal element on the lower of the two diagonals treated by a cell reaches it, then also the element in the same column j rows above it and the proper multiplicand reaches it. We number the cells in the horizontal direction from left to right and use the' index i for these numbers. Then i-j identifies the diagonal on which the updated element falls with the convention that i-j = 0 corresponds to the main diagonal, positive values to diagonals above it and negative values to diagonals below it, *Figure 43.*

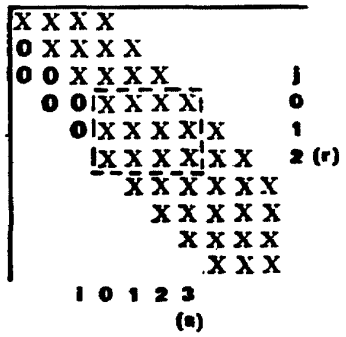


Figure 43. Indexing and distances of matrix elements.

Partial pivoting cause a performance penalty in a fully pipelined array. The utilization of the two-dimensional array decreases. It is possible to perform Gaussian elimination with partial pivoting in a linear array in the same number of cycles as in the two-dimensional arrays.

Two alternatives were presented for the treatment of systems of equations of a size larger than the array could handle in its basic form. In the first alternative the storage of intermediate results was made within the array. The intermediate storage was implemented by shift register cells. The drawback is that the register length depends on the size of the problem. It may be preferable to implement the storage as a random access memory. Partial pivoting can be accomplished as in the basic form of the array. Alternative one can be considered as an intermediate form between a purely sequential machine and a machine having an arithmetic capability associated with every storage location.

In the second alternative intermediate storage was assumed to be accomplished outside the elimination part of the array. A partitioning into three parts is effectively enforced. The parts are the elimination part, and the forward and back substitution parts, respectively. The cells had to be redesigned slightly compared to the basic array cells but are completely problem-independent. The compute time disregarding edge effects is $(3+2Sb+2Rb+2Sb*Rb)nN$, where nN is the size of the matrix and Sb and Rb are the number of nonzero blocks above and below the diagonal respectively. This represents a reduction of $O(n^2)$ compared to the sequential machine case. Partial pivoting is not easily accomplished in this alternative since only a fraction of the columns and the rows are present in the array during during any step of the computations.

The second alternative suggested for "oversized" problems is an array capable of performing matrix multiplication as well as elimination. The array can compute expressions of the form A = B + C*D in time 2n, were n is the size of the matrix.

Acknowledgement

This work has benefited from discussions with Prof. Heinz-Otto Kreiss, Applied Mathematics, Caltech, and numerous discussions with Danny Cohen, USC/Information Sciences Institute, on pipelined architectures in a joint effort on pipelined arrays for the computation of the Discrete and Fast Fourier Transforms, which is gratefully acknowledged.

This work has been made possible through the support of the Defense Advanced Research Project Agency under contract N00014-79-C-0597.

References

1. Cohen Danny, "Mathematical Approach to Computational Iterative Networks," *Proceedings of the Fourth Symposium on Computer Arithmetic*, pp. 226 - 238, October 1978, also published as USC/Information Sciences Institute RR-78-73, November 1978.

2. Johnsson Lennart, Uri Weiser, Danny Cohen and Alan Davis, "Towards a Formal Treatment of VLSI Arrays," *Proceedings of the Second Caltech Conference of VLSI*.

3. Johnsson, Lennart, "Gaussian Elimination and Concurrency: A Complexity Analysis," Internal report 4087, Computer Science, Caltech, December 1980.

4. Kung H T and Charles E. Leiserson, "Algorithms for VLSI Processor Arrays," chapter 8.3 in Carver A. Mead and Lynn A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

5. Kung S Y, Lecture Notes, Caltech, 1980.

6. Weiser Uri and Alan Davis, "Mathematical Representation for VLSI Arrays," University of Utah, Computer Science Department, Report UUCS-80-111, September 1980.